View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UT Digital Repository

Copyright

by

Cameron Taylor Bielstein

2015

The Thesis Committee for Cameron Taylor Bielstein

Certifies that this is the approved version of the following thesis:

UbiPAL: Secure Messaging and Access Control for Ubiquitous Computing

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor:

Lorenzo Alvisi

Co-Supervisor:

Robert F. Dickerson

UbiPAL: Secure Messaging and Access Control for Ubiquitous Computing

by

Cameron Taylor Bielstein, B.S.C.S

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2015

Dedication

Dedicated to my parents, Chris and Shelley, who taught me to always reach for the next challenge.

Acknowledgements

I would like to acknowledge the abundance of help and guidance from my thesis advisor, Dr. Robert F. Dickerson, who stepped in when I was in need of help and provided new direction to this project. I would also like to acknowledge the mentoring of Dr. Ahmed Gheith who taught me to believe I can make a positive contribution to academic research. Finally, thanks goes to Lori McNabb, Tiffany Buckley, and Dr. Lorenzo Alvisi for going above and beyond to provide me with this opportunity.

Abstract

UbiPAL: Secure Messaging and Access Control for Ubiquitous Computing

Cameron Taylor Bielstein, MSCompSci

The University of Texas at Austin, 2015

Supervisor: Lorenzo Alvisi

Co-Supervisor: Robert F. Dickerson

The ubiquitous computing environment and modern trends in personal computing, such as body sensor networks and smart houses, create unique challenges in privacy and access control. Lack of centralized computing and the dynamic nature of human environments and access rules render most access control systems insufficient for this new category of systems. UbiPAL is an object-oriented communication framework for ubiquitous systems which provides secure communication and decentralized access control. UbiPAL uses a modified SecPAL implementation to provide reliable, ad hoc access control. The UbiPAL system uses cryptographically signed, publicly held namespace certificates and access control lists in the style of TLS certificates. This approach allows message authentication and authorization in an ad hoc, completely decentralized method while maintaining human readability of policy language. UbiPAL was implemented as a C++ library, made freely available at [1], and evaluated to have minimized overhead. Even on the slowest device evaluated, a Raspberry Pi, UbiPAL authentication and authorization adds less than 20 milliseconds to the delivery a message with a message overhead of 153 bytes. The UbiPAL programming model separates access policy from application programming and results in small amounts of code required from the application programmer, creating an accessible paradigm for programming ubiquitous computing systems.

Table of Contents

List of Tables viii

List of Figures ix

List of Equations x

1. Introduction 1

2. Background 4

3. UbiPAL 8

   3.1 UbiPAL system 8

      3.1.1 Distributed Namespace 9

      3.1.2 Ad Hoc Access Control 10

   3.2 UbiPAL Language 12

4. Implementation 15

   4.1 Events, Messages, and Threads 15

   4.2 Message Privacy and Authentication 17

   4.3 Message Delivery: Polling vs. Pushing 17

5. Evaluation 19

   5.1 Case Studies 19

      5.1.1 House Guest Example 19

      5.1.2 Medical Heart Rate Monitoring 21

   5.2 Overhead 26

      5.2.2 Network Overhead 30

6. Future Work 35

7. Related Works 37

8. Conclusion 40

Works Cited 41

Vita 46

List of Tables

Table 1: Systems used for evaluation. 27

List of Figures

Figure 1: Assuming the existence of a process named Cluster that allows command dbgrep and a secure token service STS, these SecPAL assertions allow Alice to run dbgrep on Cluster. 5
Figure 2: UbiPAL namespace certificate.....10
Figure 3: UbiPAL access control list. 11
Figure 4: UbiPAL additions to the SecPAL scheme. Bolded elements have been added for UbiPAL. 12
Figure 5: Access Control List tree for the Bob's House example from below. Bob's door is the root, delegation goes through Bob to Alice, collecting the BOB_IS_HOME condition. 16
Figure 6: bob_door's access control list delegates to bob the authority to allow individuals to open the door. Bob is not allowed to re-delegate. 19
Figure 7: Bob's access control list says Alice may open the door only if bob_house confirms he is home. 20
Figure 8: bob_house's access control list restricts access to Bob's presence information to only elements of the house, then defines bob_door as an element of the house. Access control on confirmation messages reduces the risk of leaking information. 20
Figure 9: Message flow and delegation in the Bob's house example. 20
Figure 10: Message flow and delegation in the Chris' heart rate monitor example. 22
Figure 11: Access control list for Chris' heart rate monitor. 22
Figure 12: Access control list for Chris' smartwatch. 22
Figure 13: Access control list for Dr. Bob. 23
Figure 14: Access control list for Dr. Alice. 23
Figure 15: Access control list for Chris' smartphone. 23
Figure 16: Pseudo code for Chris' heart rate monitor. 24
Figure 17: Pseudo code for main and message RequestHeartRate response/update handler for Chris' smartphone. 24
Figure 18: Pseudo code for main and message Alert handler for Chris' smartwatch. 24
Figure 19: Pseudo code for main and message RequestHeartRate response/update handler for Dr. Alice. 24
Figure 20: Pseudo code for main and message handlers for Dr. Bob. 25
Figure 21: Computational Time of RSA and AES operations in milliseconds against a log scale. RSA key generation is excluded as it only happens once per service and cannot be optimized out. Each data point is average of 1000 operations. 26
Figure 22: Comparison of UbiPAL's send message function and standard TCP Unicast in milliseconds. Each data point is the average of 80 and 91 operations for SendMessage and TCP Unicast, respectively. 29
Figure 23: Comparison of UbiPAL send, receive, and evaluate ACL statement in milliseconds. Each data point is the average of over 91, 93, and 80 for send, receive, and evaluate statement, respectively. 30
Figure 24: Network byte layout of a UbiPAL namespace certificate. 32
Figure 25: Network byte layout of a UbiPAL access control list. 32
Figure 26: Network byte layout of a UbiPAL message. 32

List of Equations

Equation 1 $Messaes\ when\ polling = 2 * frequencypoll * time$ 33
Equation 2 $Messages\ with\ pushed\ updates = 2 + (time * \ frequencyupdates)$ 33

1. Introduction

Recent trends in computing devices are fueling a drastic change in the computing landscape. Developers are presented with an increasing number of cheap, low-end devices for embedded computing with popular examples found in devices such as the Raspberry Pi and the BeagleBone Black. The availability of these low-cost, physically small computing devices has given rise to the design of "smart" devices. These devices take their so-called intelligence from embedded computer systems that infuse objects typically considered outside the realm of computing with the advantages of network connectivity and programmable logic. The increasing complexity and number of these devices has presented the opportunity for the devices to network and collaborate to achieve user objectives in a real-world, human environment. Smarthomes, sensor networks, and wearable technology are three emerging categories of distributed systems.

These new models of distributed systems bring with them a unique challenge in privacy and access control. Increasing ubiquity of devices means an increasing integration with the user's environment and potentially increased availability of sensitive user information or control over user safety. By nature, some of these devices may be responsible for privacy and safety concerns of the users through the control of security devices such as locking doors and security cameras or medical devices including pacemakers and heart rate monitors. Access control and privacy in this context becomes increasingly challenging as a smarthome may need to differentiate homeowner from house guest and change behavior when a guest is present. Since ubiquitous systems are integrated into human environments, users may play multiple roles, perhaps simultaneously. Personal and professional roles carry privileges to different resources and multiple users can be present in a given situation with ranging levels of privileges. Furthermore, access control requirements in real-world scenarios may involve control of access to physical entities that may

not be standard across all users. Technical challenges abound in the ubiquitous world as well. There are no guarantees of availability in such a real-world scenario. Systems in the network may have intermittent network connections, extremely low-powered processing capabilities, or may be physically mobile.

Since these ubiquitous systems reside as elements of the human environment and may act at the request and on the behalf of human users with real world results, it is beneficial to consider access control from a human perspective. Human decisions of access control may be strongly circumstantial. Human access control decisions can be as trivial as a stranger asking for the time of day or a mutual friend asking for admission to your home. While the prior decision may be simple, the latter decision is much more nuanced. Such a decision may depend on if your mutual friend is present and willing to vouch for this individual, the lateness of the hour, or the presence of severe weather outside of your home.

This paper seeks a digital expression of such complex, conditional access control decisions that is protective of user privacy and resilient against network partitions and low availability. This paper builds such a system through extension and modification of the SecPAL policy assertion language [2]. The major contributions of this paper are as follows:

- Extension of the SecPAL language to allow for policy assertions with fully dynamic external conditional statements.
- Design of a modified SecPAL system using publicly held, cryptographically signed namespace certificates and access control lists to remove any single point of failure, such as reliance on a centralized certificate authority.

- Implementation of a software library which embodies the above while focusing on ease of programmability for the application developer. The source code of this library is provided open and free for use by the ubiquitous computing community.

2. Background

Although several successful access control solutions exist, they do not rise to the challenge of a dynamic, real-world environment. A few relevant examples are presented below.

Readers may be familiar with POSIX access control lists (ACLs) [3] from their prevalent use in POSIX-compliant operating systems. POSIX ACLs are a textbook definition of discretionary access control (DAC). DAC is a method of access control in which certain users may pass access permission either directly or indirectly on to a third party. POSIX ACLs are implemented with the familiar owner, group, and others permissions for read, write, and execute. In this system, the file owner may allow or disallow access to specific groups but not have exclusive control over the membership of those groups. In this way, access permission may be delegated to users with permissions to administrate user groups. POSIX ACLs have seen success in single-machine systems, however such control lists fail to express the complex rules that are required to operate in a human space. They suffer from strict centralization and high administrative overhead. All authorization and authentication is performed by the local kernel. Such high centralization does not scale well to the external world. In a complex system, minor changes to desired user abilities may require refactoring of user groups and tedious permission changes on individual files. Furthermore, basic access is controlled over a predefined set of actions: read, write, and execute. More fine-grained levels of control, such as allowing append but not modification or write without deletion, would require changes of the underlying operating system in order to implement. Additionally, once set, an access control list is valid under all circumstances until manually updated again. Such a static system is not well suited to the challenges of a dynamic human environment.

More distributed approaches to access control have been attempted. Kerberos [4] is a well-known centralized system for network authentication in a client-server format. In order for two clients to authenticate themselves with each other, the clients first authenticate themselves with the Kerberos authentication server. The authentication server challenges against the private key of the client and presents each client which passes the challenge with a ticket, which the clients use to prove their identities to each other. Although Kerberos was originally based solely on symmetric cryptography [4], later revisions allowed initial client authorization using asymmetric cryptography with public keys [5]. Although Kerberos allows for reliable authentication of clients, it does not access control or request authorization. The system also requires all participants to be registered with the system, unknown clients may not connect, and has a strong single point of failure. These three restrictions hold Kerberos back from being the full answer to the access control in the ubiquitous computing environment.

A different direction has been proposed by Becker, Fournet, and Gordon in their research on SecPAL [2, 6]. SecPAL (Security Policy Assertion Language) is a DAC access control language which aims to decentralize the creation and administration of access control lists and the authorization of requested operations in distributed systems. Their research describes both an access control language and a system for that language. The SecPAL system describes a set of SecPAL policy assertions which combine to allow access to appropriate parties. These assertions may directly allow access to a user or class of users, classify users, or delegate the ability to make assertions about specific resources. SecPAL allows some dynamic checks such as temporal access constraints. Figure 1 shows a simple example from [2] with rules necessary to allow user

```
Cluster says STS can say X is a
        researcher
Cluster says X can execute dbgrep if
        X is a researcher
STS says Alice is a researcher
```

Figure 1: Assuming the existence of a process named Cluster that allows command dbgrep and a secure token service STS, these SecPAL assertions allow Alice to run dbgrep on Cluster.

Alice to execute command dbgrep on Cluster. These rules assert that researchers are able to run the application dbgrep and that Alice is a researcher, as stated by STS on the authority of Cluster. Therefore, Alice can execute dbgrep. This assumes the existence of a process named Cluster and a secure token server called STS.

These assertions, or tokens, are held at the node which makes a given assertion and would be collected by Alice and sent with her request to Cluster. Cluster would evaluate the rules based on the union of local tokens it has created as well as the tokens sent by the request. Since Cluster is the resource guard, it is responsible for checking the tokens, but Alice is responsible for gathering and sending any tokens the resource guard does not hold.

However, despite the advances for access control in a distributed environment, SecPAL is not completely appropriate for ubiquitous systems. First, SecPAL has multiple points where single-node failure or network partitioning may block system execution. Such a weak liveness property is undesirable in a distributed system where node availability is not guaranteed. There are two major examples of single-node failure in SecPAL. The first is revealed in the necessity of a central external secure token authority. The SecPAL implementation discussed by Becker, Fournet, and Gordon is built on top of Kerberos [2]. If this token authority fails or becomes partitioned from parts of the network, any node which is unable to reach the Kerberos server will be unable to authenticate and execution will be blocked until contact with the Kerberos server may be regained. The second point of failure is through credential gathering [7]. Credentials are stored locally at each node in the SecPAL network and can be gathered through a method presented by Becker, Mackay, and Dillaway [7]. Becker, Mackay, and Dillaway acknowledge that if a node holds some of the credentials required and becomes unavailable to the network for

any reason, credential gathering cannot successfully gather all credentials needed to pass to the resource guard and will fail.

The SecPAL language is also not fully equipped for real-world access control in that the language does not allow fully dynamic conditions based on real-world conditions. Although SecPAL does include some dynamic conditions, such as checks of file attributes and temporal conditions [2], external conditions are not considered part of the access control. Consider the previously described house guest example, it would be impossible to allow the individual into the house on the condition of inclement weather based purely on SecPAL security assertions. It would be a requirement for a user application to do this on its own, which divides static and dynamic access control elements between various programming elements: SecPAL and the application running on SecPAL. This is undesirable as it places more implementation responsibility on the shoulders of the application developer and increases complexity of the security model. Such a model would also be at increased risk of malicious applications.

3. UbiPAL

This section describes the design of UbiPAL, the Ubiquitous Policy Assertion Language. UbiPAL is a SecPAL-based access control system designed to address the shortcomings of SecPAL as described above. UbiPAL, and thus this section, is divided into two parts: the UbiPAL system and the UbiPAL language. The UbiPAL system is a message passing system responsible for secure inter-process communication over a network. It includes a completely decentralized ad hoc SecPAL implementation based on the UbiPAL language for access control. The UbiPAL language is an extended version of SecPAL to support human-readable policy assertions with external conditions.

3.1 UbiPAL system

The UbiPAL system is designed as disjoint processes communicating through authenticated and secure network messages. The model follows the design pattern of the Mach microkernel [7] of modeling networked machines as objects between which messages are passed to trigger events. As Mach maps port numbers to processes, UbiPAL maps UbiPAL names to services. Access control is specified on the types of messages each service may receive from a given sender. Each message is defined as a message title and message arguments, much like a function call. The receiving service only delivers the message if the sending service successfully passes authentication and has the authority to send that type of message to the receiving service.

To be completely decentralized, message passing and message authorization must happen without access to any specific service outside of the sender and receiver. Each service should be able to make its own decisions on access control and services must be able to communicate directly to each other without any routing service or namespace server. Failure to meet such requirements introduces a single point of failure to the network. The potentially mobile nature of

ubiquitous computing makes network partitions inevitable and failure to handle those partitions undesirable. To achieve these goals, UbiPAL uses a distributed namespace with ad hoc access control.

3.1.1 Distributed Namespace

The UbiPAL namespace is a flat network of services. A service is an abstraction for a node on the network and services need not have a one-to-one mapping to a physical machine. A single machine may run may services or a service may actually represent many machines. A service name may even represent a user.

The services are authenticated based on an asymmetric, cryptographic public key infrastructure similar to the Transport Layer Security (TLS) protocol [8], or its predecessor the Secure Sockets Layer (SSL) protocol [9]. Services are uniquely identified solely by their public key. This idea, presented by Blaze, Feigenbaum, and Lacy in 1996 [10] and used by the Simple Public Key Architecture (SPKI) [11] and the SecPAL implementation from [2] removes the layer of indirection created through a mapping of name to public key in systems such as TLS. This works because a soundly generated private key generates a globally unique public key [12]. The authors of the SPKI Certificate Theory report for the Internet Engineering Task Force note that in addition to globally unique identifiers, the use of public keys for identifiers makes more sense in the digital world since there is a small amount of information gleaned from an arbitrary name for a computer, despite humans placing huge importance on names in trust-based relationship decisions [12]. Furthermore, this simplifies challenging the identity of a service as the name can be used as a public key to encrypt a challenge message or check a message signature. In this way, the only vital piece of information for each service is the private key. If the private key is saved to

a file or transferred securely between devices, the service may restart after system failure or migrate between devices.

Once a process generates its public and private key pair, it creates a UbiPAL namespace certificate. The namespace certificate is the public key, also known as the service identifier, an Internet Protocol (IP) address and port number to which messages for the service should be sent, and a certificate version number. The version number is a monotonically increasing non-negative integer used to differentiate more recent versions of the namespace certificate in the event of updates. This update mechanism promotes simple updates in the face of device mobility. The certificate is signed with the service's private key. The inclusion of both the public key, as the service identifier, and a private key signature of the certificate creates tamper-free certificates and allows each certificate to be authenticated without a central certificate authority.

```
Service Identifier (Public key)
Internet Protocol address
Port number
Version number
Private key signature
```

Figure 2: UbiPAL namespace certificate.

Because the certificates are self-authenticating, namespace certificates may be cached at each service and forwarded as is to requesting services to serve as an ad hoc namespace lookup service. Neighboring services may be polled for a given namespace certificate until it is found. Once a certificate is found mapping the desired service identifier to an IP address and port, the services may communicate directly to each other.

3.1.2 Ad Hoc Access Control

SecPAL stores policy assertion credentials on each node and checks the credentials at the receiving service [2] and the SecPAL research group presented a method of gathering those credentials [7]. However, their method of abductive credential gathering [7] does not allow for a

node holding required credentials to fail. The UbiPAL system solves this problem through the creation of publicly held access control lists.

Each element collects access control rules into access control lists. They may be grouped as a single list or separated between multiple lists for fine-grained caching and forwarding by other services. These access control lists, layout shown in Figure 3, are signed with the private key of the service for verification and broadcast to neighbors in the network. In the same manner as namespace certificates, UbiPAL access control lists are self-authenticating and resistant to tampering. Therefore, these rules can be sent publicly or privately to neighbors in the network to preserve privacy, and non-private access control lists can be forwarded arbitrarily by other services in the network. Upon receipt of a message, the receiving service should evaluate the right of the sending service to send that message based on the non-revoked access control lists heard by the receiving service at the time of message receipt. Should the current access control lists known by the receiving service not allow the message to be delivered, the receiving service drops the message and notifies the sending service of the unauthorized status of the message. Although this service may produce false negatives, as there may exist an access control list allowing the message that the receiving service has yet to receive, it will result in no false positives, which must be avoided for secure access control. To mitigate a false negative, the sending service may forward any additional access control lists to help fill gaps in the receiving service's access control lists and retry the message, if desired.

```
Service Identifier (Public key)
Globally unique access control
      list identifier
Private/Public
Number of rules (Num_Rules)
Num_Rules times: rule
```

Figure 3: UbiPAL access control list.

3.2 UbiPAL Language

Due to its design, SecPAL is naturally extensible [2, 6]. For the ubiquitous computing context, SecPAL must be extended to control access of messages and to accept dynamic external conditions. The SecPAL language is simply a set of assertions about access control. Each assertion is comprised of an *issuer*, a set of *conditional facts*, and a *constraint* [2]. The UbiPAL language defines all access control in terms of the public key service identifier.

SecPAL allows declaration of user-defined predicates, which UbiPAL leverages to introduce the "can send" predicate to describe message sending rights. The predicate has arity two and is the following syntax: can send [-] to [-]. This predicate describes an access control statement that allows the service specified by the second argument to receive the specified message. With this additional predicate, UbiPAL may express access in terms of messages passed between elements in the namespace.

SecPAL also allows the addition of constraints without affecting decidability or tractability [2]. UbiPAL adds a "confirms" condition with arity two of the following syntax: [-] confirms [-]. The first argument is a service

e :: = x	(variables)
| A	(constants)
***pred* ::= can send message [-] to [-]**	**(message control)**
***D* ::= non-negative integer**	**(re-delegation length)**
verbphrase ::= *pred* $e_1 \ldots e_n$	*for n = Arity(pred)*
| can say [D] *fact*	**(delegation)**
| is *e*	**(aliasing)**
| is a *e*	(grouping)
fact ::= *e verbphrase*	
***op* ::= <**	**(less than)**
| >	**(greater than)**
cond ::= *cond* $e_1 \ldots e_n$	*for n = Arity(cond)*
| [-] confirms [-]	**(remote condition)**
| CurrentTime() *op e*	**(temporal condition - time)**
| CurrentDate() *op e*	**(temporal condition - date)**
| *pred*	
rule ::= e says fact	(assertion)
| *e says fact if* $cond_1, \ldots, cond_n$	(conditional assertion)

Figure 4: UbiPAL additions to the SecPAL scheme. Bolded elements have been added for UbiPAL.

identifier and the second is a message to send. The condition will return true if the evaluating service has a namespace certificate for the given service identifier, the evaluating service is privileged to send the given message, the confirming message responds with a defined confirmation message and, to ensure liveness, the entire evaluation does not surpass a configurable timeout period. This confirmation allows UbiPAL rules to be written to adjust to human environmental factors as detected by remote services. Other services in the system may provide information through the confirms condition to allow a service to make a dynamic decision on access control. Since confirms messages are subject to all the rules of access control and are answered only with a confirmation or error message, user privacy is not compromised.

UbiPAL also introduces simplified depth of delegation. SecPAL allows the argument D in the delegation verbphrase to be 0 or ∞, representing no re-delegation or unbounded re-delegation. The authors of SecPAL point out that chaining can say[0] N times allows N levels of delegation for any non-negative integer N [2]. However, this syntax quickly becomes verbose. UbiPAL simply defines the argument D to be any non-negative integer, creating a much cleaner syntax.

Because UbiPAL language specifies all names as the public keys, policy rules can get quite long. The UbiPAL policy assertion language introduces the predicate “is” in addition to “is a.” “is” can be used to shorten existing statements as shown in some of the examples in Section 5.1. Finally, UbiPAL loosens assertion safety requirements from SecPAL. SecPAL requires any variable mentioned in the assertion to also be listed in the conditional facts [2]. The authors of SecPAL assert this safety requirement ensures completeness and termination of their evaluations, which are run after being translated into Datalog. Although this does not fully preclude blanket access control rules, it does increase the complexity as users and clients must declare themselves as a member of some group [2] and therefore must have a knowledge of the system. UbiPAL

changes the method of evaluation and needs not make such a constraint on variables, allowing true global access assertions.

4. Implementation

The UbiPAL specification as stated above was implemented as an open source C++ library. The library is available for review and modification at [1]. Some important implementation details are discussed in this section.

4.1 Events, Messages, and Threads

The UbiPAL library is a multi-threaded, event-driven library. When a service is executed, it starts several threads for sending, receiving, and handling of messages. Upon receiving a message, the message receive event is triggered. This event authenticates the sender, authorizes the message being received, and handles the message as appropriate by type.

Messages on the network are passed as three basic types: namespace certificates, access control lists, and messages. Namespace certificates and access control lists are handled by UbiPAL code without need for application programmer defined handlers. Most messages are handled by application programmer code. Some specific messages are handled by UbiPAL, such as requests for specific namespace certificates or access control lists.

For received messages not handled by the UbiPAL library code, the application programmer must register callback functions for each message type the service will handle. Upon receiving an authorized, authenticated message of a type with a registered callback, the UbiPAL library calls that function, effectively transferring control of execution back to the application programmer and executing the user-defined code for that message.

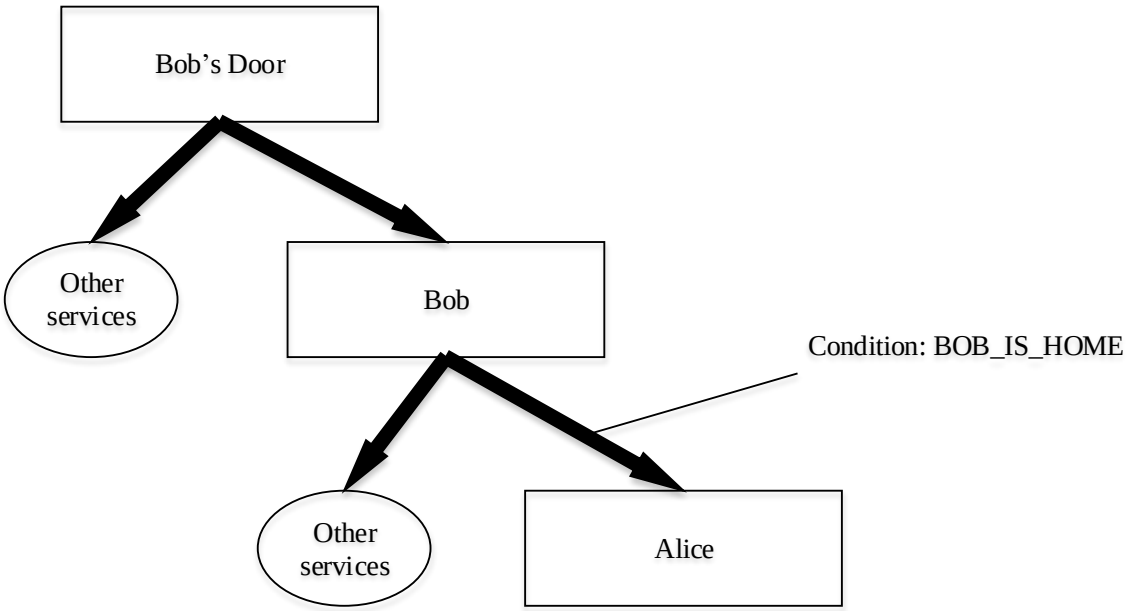

Figure 5: Access Control List tree for the Bob's House example from below. Bob's door is the root, delegation goes through Bob to Alice, collecting the BOB_IS_HOME condition.

Message authorization is performed as a bounded, breadth-first search of the tree of access control lists on hand. Using a tree search allows for a search of the available access control rules, which may be rapidly changing, in linear time. This is preferable to SecPAL authorization queries, which take polynomial time [2]. An authorization tree is built relative to the resource guard which, when a message is being received, is the receiving service. Services are the nodes in the authorization tree while the resource guard is the root. Links in the tree are created by delegation of the fact in question. The authorization process searches down the tree following the delegation links to the maximum depth specified by the "can say" verbphrase until either a policy assertion allowing the requested operation is found and the request is authorized or until the tree search is exhausted and the request is not authorized. Conditions are collected along the path and evaluated when a path successfully terminates. Lazy condition evaluation saves the overhead of sending unnecessary confirmation messages.

4.2 Message Privacy and Authentication

Message privacy is ensured through encryption. Messages are sent between services with RSA [13] public keys for service identifiers. Since identifiers are RSA public keys, the identifier of the receiving service can be used to encrypt a message that can only be decrypted by the receiving service, which holds the matching private key. This guarantees the receiving service is the intended service, even if the address or port used actually maps to a different service as the other service will not be able to decrypt the message without the private key of the destination service. To authenticate the sender, each message is signed by the private key of the sending service. Since the message is sent with the sending service's identifier, which is the public key, the identity of the sender and the integrity of the content can be authenticated based on message content alone.

However, the UbiPAL library makes as sparing use of RSA encryption as possible. As discussed in the evaluation section below, RSA asymmetric encryption is much less performant than symmetric encryption. Therefore, UbiPAL, on first contact with a new service, exchanges Advanced Encryption Standard (AES) [14] symmetric private keys to use for all future encryption between the two services. This dramatically reduces the overhead of encrypted communication.

4.3 Message Delivery: Polling vs. Pushing

The object-oriented nature of UbiPAL lends itself to a polling method of data acquisition. A service requiring data from a neighboring service simply sends a message requesting the data and awaits the reply. In some scenarios, discussed below in the evaluation, this is not an efficient use of network resources. Instead, to avoid excess network traffic, the neighboring service should be able to notify the requesting service when new data is available. UbiPAL accomplishes this

through message caching, which is this system's take on the classic computer science tradeoff of polling versus pushing. Rather than writing custom message handlers, UbiPAL programmers may register their current response for a given message with the UbiPAL service. Remote services may register to receive updates on that message response and when a new response is set, each registered service will receive an update message. The registered service caches that updated response on its local machine. That message cache is checked before any outgoing message is sent. If there is a response in the cache for the outgoing message, the outgoing message is not sent and the cached response is delivered to the given handler immediately. In certain situations, discussed below in the evaluation, this push model can save significant amounts of network overhead.

5. Evaluation

This section presents an evaluation of the UbiPAL language and system, as implemented in the UbiPAL library described above. Case studies are presented to demonstrate the ease of programming in the UbiPAL model and evaluation of the system's overhead are presented to demonstrate the minimal performance tradeoff necessary to program inside the UbiPAL model.

5.1 Case Studies

Two example case studies are presented below. The first example is very straight forward and included primarily to aid the understanding of UbiPAL access control lists. The second scenario is a bit more involved and includes pseudo code and full access control lists for each service in the scenario.

5.1.1 House Guest Example

A simple example is given of an interaction in UbiPAL. The next subsection will present a more complex scenario. This scenario describes Alice visiting Bob's home. Bob would like his door to unlock for Alice, but only if he is home. This example will use English names for service identifiers rather than public keys, for the sake of readability. This example assumes the following services and users exist *alice, bob, bob_door, bob_house*. The access control lists for each device are listed in the figures below.

```
bob can say[1] X can send OPEN to bob_door
```

Figure 6: bob_door's access control list delegates to bob the authority to allow individuals to open the door. Bob is not allowed to re-delegate.

```
alice can send OPEN to bob_door if bob_house confirms BOB_IS_HOME
```

Figure 7: Bob's access control list says Alice may open the door only if bob_house confirms he is home.

```
X can send BOB_IS_HOME if X is a house_element
bob_door is a house_element
```

Figure 8: bob_house's access control list restricts access to Bob's presence information to only elements of the house, then defines bob_door as an element of the house. Access control on confirmation messages reduces the risk of leaking information.

When Alice arrives at Bob's house, she can send a message to the door, perhaps with a smartphone, to request that the door unlock for her by sending OPEN to *bob_door*. Since *bob_door* has delegated access control to *bob*, *bob*'s access control list comes into play, which gives the conditional statement that Alice may open the door if Bob is home. *bob_house* allows any member of the house_element group to check for presence in the house and specifies *bob_door* is an element of the house. This allows *bob_door* to confirm BOB_IS_HOME. With these rules, if Bob is home, Alice is allowed to unlock the door. Such a situationally aware

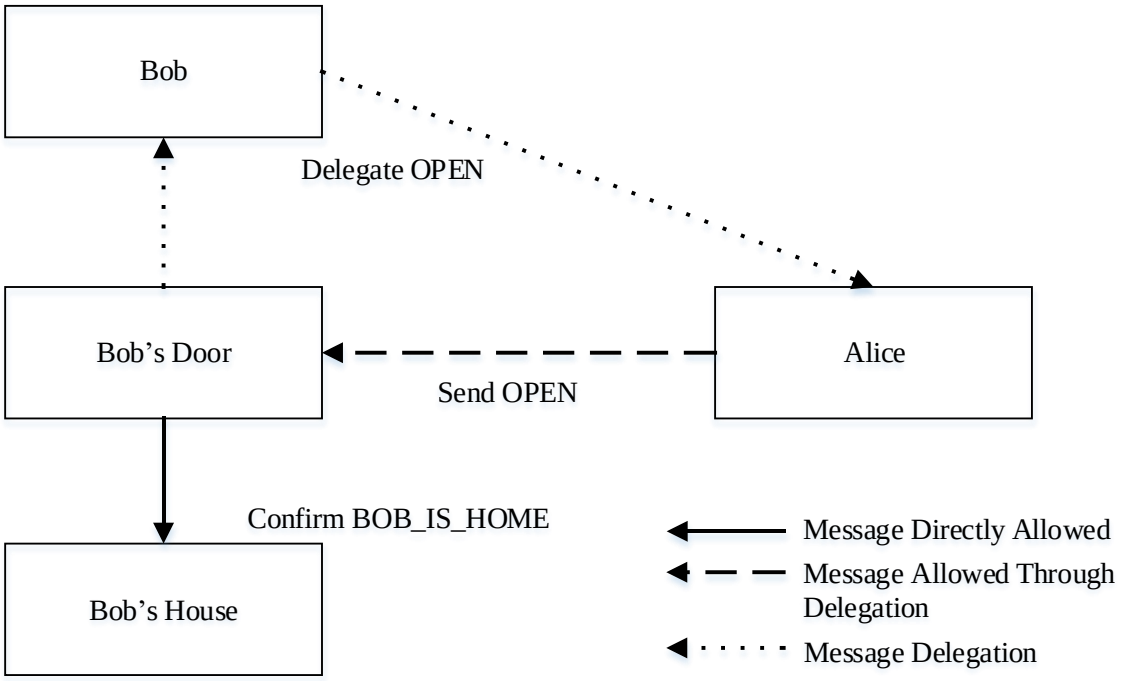

Figure 9: Message flow and delegation in the Bob's house example.

scenario is not possible in SecPAL. Without the confirms conditional keyword, SecPAL would require moving parts of this access control logic into the application code and out of the policy assertions. This is also completed without the need for a central token server login before the interaction. Each service, upon encountering each other for the first time, would simply swap namespace certificates and, optionally, access control lists.

5.1.2 Medical Heart Rate Monitoring

This example is slightly more involved. There are more actors involved in more complex interactions. Dr. Alice is a general care doctor who works with heart specialist Dr. Bob. Drs. Alice and Bob are caring for their patient, Chris, who has a heart condition. Chris has been instructed to wear a heart rate monitor at all times to report information about his heart back to Dr. Alice in her office. If the heart rate is outside of a normal range, for this example defined to be below 60 beats per minute or above 120 beats per minute, Dr. Alice will page Dr. Bob with the information, who may then request heart rate information directly from Chris' heart rate monitor. Chris will also be notified on his smartwatch when his heart rate is outside of normal bounds. Chris' smartphone has an app to allow him to further share his heart rate information, which

includes the option to temporarily disable sharing by turning off the share heart rate setting through an app on his smartwatch. The desired interaction is illustrated in Figure 10 below.

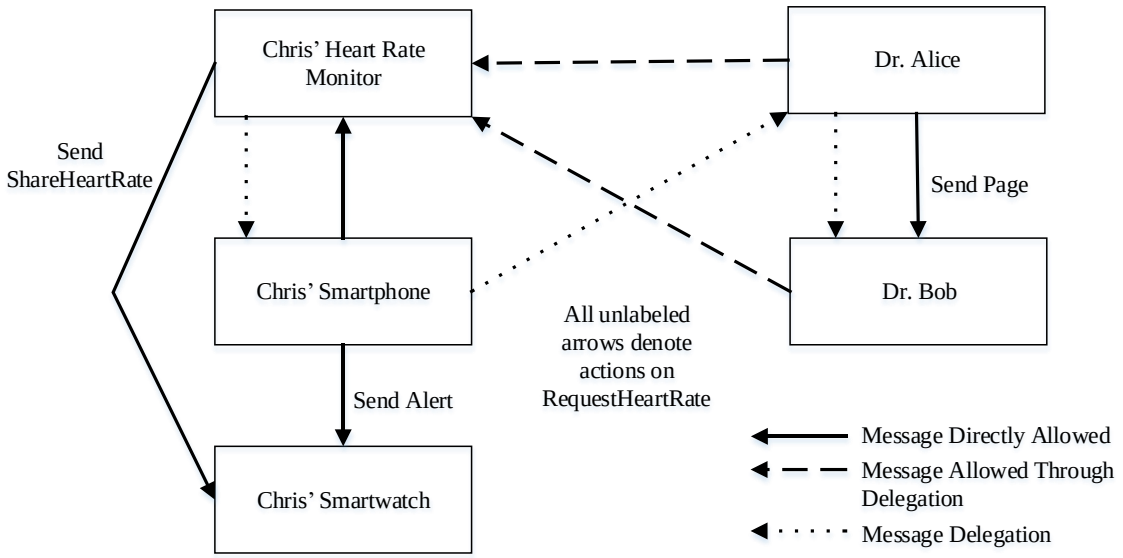

Figure 10: Message flow and delegation in the Chris' heart rate monitor example.

The access control lists for each service in this interaction are listed below. These are taken directly from the implementation used for testing. Note that each access control list is human readable and no more than a few lines in length each. Public key identifiers have been shortened for readability.

```
CDEB… IS smartphone
CC89… IS hrm
X CAN SAY Y CAN SEND MESSAGE RequestHeartRate TO Z if X IS
        smartphone, Z IS hrm
X CAN SEND MESSAGE RequestHeartRate TO Z if X IS smartphone, Z IS hrm
```

Figure 11: Access control list for Chris' heart rate monitor.

```
CDEB… IS smartphone
E46F… IS smartwatch
CC89… IS hrm
X CAN SEND MESSAGE ShareHeartRate TO Z if X IS hrm, Z IS smartwatch
X CAN SEND MESSAGE Alert TO Z if X is smartphone, Z IS smartwatch
```

Figure 12: Access control list for Chris' smartwatch.

```
AE24… IS DrAlice
C39E… IS DrBob
X CAN SEND MESSAGE Page TO Y if X IS DrAlice, Y IS DrBob
```

Figure 13: Access control list for Dr. Bob.

```
C39E… IS DrBob
CC89… IS ChrisHrm
X CAN SEND MESSAGE RequestHeartRate TO Y if X IS DrBob, Y IS ChrisHrm
X IS A doctor if X IS DrBob
```

Figure 14: Access control list for Dr. Alice.

```
E46F… IS smartwatch
CC89… IS hrm
AE24… IS DrAlice
X CAN SEND MESSAGE RequestHeartRate TO Y if X IS smartwatch, Y IS hrm
X CAN SEND MESSAGE Alert TO Y, X IS hrm, Y IS smartwatch
X CAN SEND MESSAGE RequestHeartRate TO Y if X IS A doctor, Y IS hrm,
        E46F… CONFIRMS ShareHeartRate
X IS A doctor if X IS DrAlice
X CAN SAY[1] Y IS A doctor if X IS DrAlice
X CAN SEND MESSAGE Alert TO Y if X IS hrm, Y IS smartwatch
```

Figure 15: Access control list for Chris' smartphone.

Listed below is the pseudo code for the interaction on each service. In most cases, the pseudo code for each service is actually shorter than the access control list. This demonstrates the power of separation of functionality and access control in the UbiPAL model. It can also be seen that although all access to RequestHeartRate messages sent to the heart rate monitor are delegated by Chris' smartphone, requests are directed to the heart rate monitor itself. The heart rate monitor evaluates requests based on the set of all access control lists it has cached. On the condition that the smartphone's access control lists have been sent to the heart rate monitor, which is necessary for any RequestHeartRate message to be delivered, then Drs. Alice and Bob could continue to receive information from the heart rate monitor even if the smartphone disconnected from the network due to low battery, for example. Under message caching conditions, this scenario can still run even with the smartwatch failing. This shows the power of UbiPAL's publicly held information. Once the relevant access control lists have been created and shared and any

conditional values have been cached, the only two nodes required for communication are the resource and the requester. All other required values can be held by those services and securely evaluated against for access control.

```
main
        loop forever
                value := ReadHeartRate()
                Set Message Reply for message RequestHeartRate to
                        value
```

Figure 16: Pseudo code for Chris' heart rate monitor.

```
main
        Register with Chris' heart rate monitor for message
                RequestHeartRate
```

```
Handle reply to message RequestHeartRate
        if message.heart_rate is not in range 60-120
                Send message Alert to Chris' smartwatch with heart
                        rate data
```

Figure 17: Pseudo code for main and message RequestHeartRate response/update handler for Chris' smartphone.

```
Main
        Register callback function "Handle message Alert" for receive
                of message Alert
        if user sets Share Heart Rate
                Set Message Reply for message ShareHeartRate to
                        CONFIRM
        if user sets Do Not Share Heart Rate
                Set Message Reply for message ShareHeartRate to DENY
```

```
Handle message Alert
        Display alert
```

Figure 18: Pseudo code for main and message Alert handler for Chris' smartwatch.

```
Main
        Register with Chris' heart rate monitor for message
                RequestHeartRate
```

```
Handle reply/update to message RequestHeartRate
        Display heart rate
        if message.heart_rate is not in range 60-120
                Send page Alert to Dr. Bob with heart rate data
```

Figure 19: Pseudo code for main and message RequestHeartRate response/update handler for Dr. Alice.

```
Main
        Register callback function "Handle message Page" for receive
                of message Page
```

```
Handle message Page
        Display page
        Send message RequestHeartRate to Chris' heart rate monitor
```

```
Handle reply to message RequestHeartRate
        Display heart rate
```

Figure 20: Pseudo code for main and message handlers for Dr. Bob.

All presented UbiPAL rules come directly from a sample implemented in the UbiPAL library [1]. Total implementation code is minimal in length. The UbiPAL C++ library requires a small additional setup code to restore the service from a file and read the access control lists from a file, for example, each of which require one line. That code is library implementation specific and is not listed here, though examples may be found in the UbiPAL C++ repository at [1].

This same scenario implemented in unmodified SecPAL has a few undesirable results. As with the previous scenario, all involved services would be required to have an account with a known token server, implemented with a system such as Kerberos, and to use that server to establish new connections. This single point of failure is avoided in UbiPAL as services authenticate themselves and authorization is directly founded on that authentication. As previously discussed in this section, UbiPAL would be resilient to nodes in the chain of delegation failing where SecPAL would not be under its model of credential gathering [7]. Finally, as discussed elsewhere, SecPAL is unable to express the conditional verification of the watch setting share heart rate mode without incorporating access control into the application code at the requesting side, which is undesirable.

5.2 Overhead

The following performance reviews are based on the code available at [1]. The analysis will focus on both network and computational overhead of the UbiPAL library. Evaluation will focus on static evaluation of network overhead, offline testing of encryption, and an analysis of statistics from an implementation of an above example. This section will also include discussion of which steps have been taken to reduce some of the overhead of the protocol.

5.2.1 Computational Overhead

The most computationally complex operation in the UbiPAL library by far is encryption. To minimize computational overhead, UbiPAL follows the work of TLS [8] and tries to minimize the use of the asymmetric cryptography. UbiPAL uses RSA [13] for asymmetric cryptography. On first contact, RSA keys are used to exchange a symmetric key. The UbiPAL library uses the

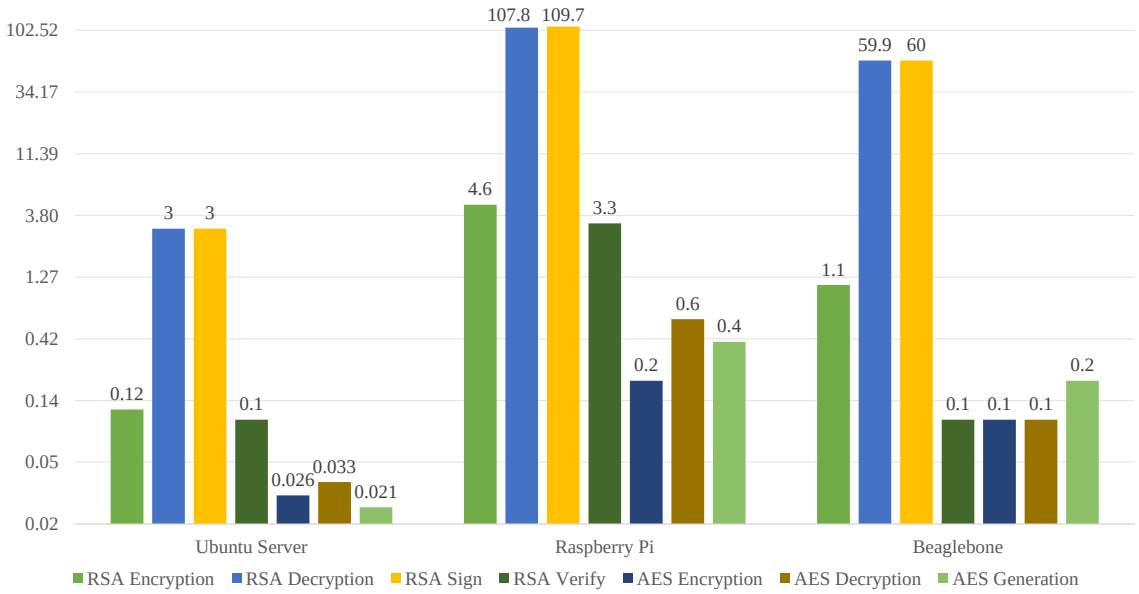

Figure 21: Computational Time of RSA and AES operations in milliseconds against a log scale. RSA key generation is excluded as it only happens once per service and cannot be optimized out. Each data point is average of 1000 operations.

AES [14] for symmetric cryptography. AES cryptographic operations are several orders of magnitude faster than the corresponding RSA cryptographic methods. Even with the additional work to generate a new AES key and IV for each remote service, there is a significant savings. Once AES keys are exchanged, the identity of both the sender and the receiver are authenticated

	Ubuntu Server	Raspberry Pi	Beaglebone Black
Processor	AMD Phenom II X4 810 2.61 GHz	ARM v6 700 MHz	ARM AM335x 1 GHz Cortex-A8
Clock Speed	2.61 GHz	700 MHz	1 GHz
Cores	4	1	1
Memory	6 Gigabytes	512 Megabytes	512 Megabytes
Operating System	Ubuntu 14.10 Server virtualized	Debian 7.8 (Raspbian)	Debian 7.8 (BeagleBone Black 2015-03-01 image)
Host System	Windows 8.1 Pro using Hyper-V 6.3.9600.17396 (virtual machine used 100% system resources, as listed above)	N/A	N/A

Table 1: Systems used for evaluation.

as those are the only two services which share the AES key used on each message, therefore eliminating the need for an RSA signature to validate the sender of the message.

The results of these changes were evaluated by finding the average type of each operation across three different devices: Raspberry Pi model b, BeagleBone Black, and a high-powered virtualized Ubuntu Server. The specifications of the systems used for test are shown in Table 1. Each operation was run 1000 times on each device on 100 randomized bytes and the averaged values are presented in Figure 21 in milliseconds against a logarithmic scale. Since RSA key generation is only run once per new service created and services may be saved and resumed from a file, it is therefore not a significant contribution to the overhead of the standard use case of UbiPAL services and is excluded from the graph.

Based on this figure, the average time for our Raspberry Pi to prepare a 100 byte message for sending, which includes an RSA signature and an RSA encryption, is 114.5 milliseconds. The same unit, on average, will take 108.5 milliseconds for the required RSA decryption and RSA verification that are required to receive that message. Using AES symmetric cryptography with the elimination of RSA signature verification, those numbers drop to a simple 0.3 milliseconds to send and 0.2 milliseconds to receive. This savings is multiplied over the total number of messages sent between two services.

A runtime evaluation was performed on the UbiPAL library. Major functions were timed to find a breakdown of the relative amount of overhead. The above heart rate monitor example was implemented in the repository at [1]. The pseudo code was translated to C++ but the access control lists were used exactly as is. Approximately 100 messages were sent and received by the five UbiPAL services involved in the example and the test was run separately on all three test systems.

Figure 23 shows the average time of the three major UbiPAL operations: message sending, message receiving, and access control statement evaluation. In this case, sending messages includes namespace certificates and access control lists, both of which always require an RSA signature, thus resulting in longer send times. These numbers are the average values over 91, 93, and 80 operations respectively and give insight into the exact computational overhead of the UbiPAL system. Figure 22 shows a comparison of sending a UbiPAL message compared to the time required to open a TCP connection and send the data without any UbiPAL operations. This figure uses only message sends and excludes namespace certificates and access control lists and operations were run 80 and 91 times, respectively. The send message operation includes message serialization, message encryption, and any necessary signature before opening a TCP connection and sending bytes. The number for TCP unicast times only the opening of the TCP connection and sending of bytes. Although the additional operations added by UbiPAL cause an increased

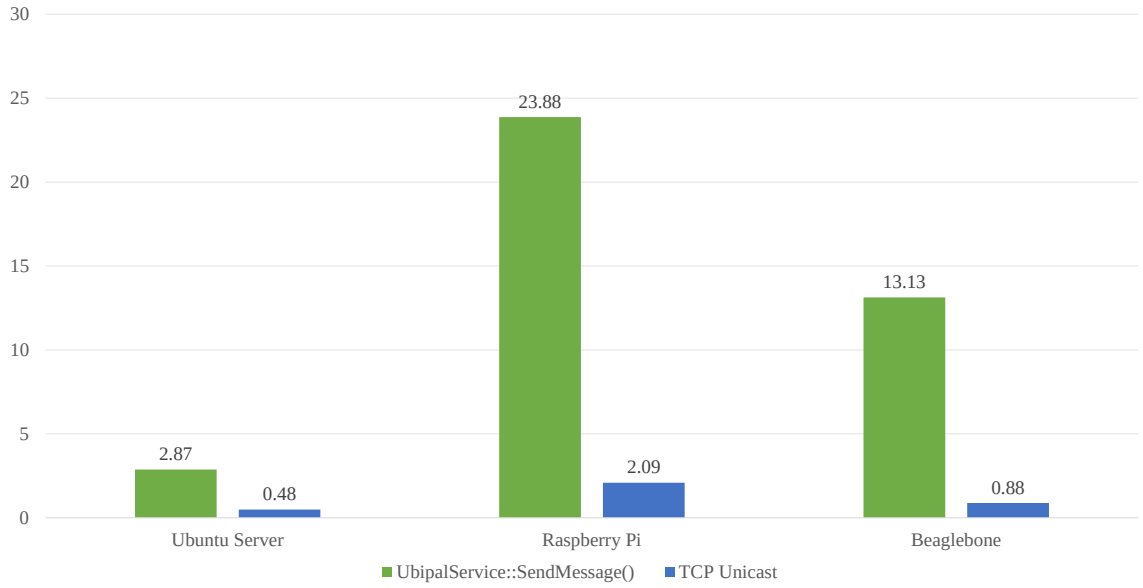

Figure 22: Comparison of UbiPAL's send message function and standard TCP Unicast in milliseconds. Each data point is the average of 80 and 91 operations for SendMessage and TCP Unicast, respectively.

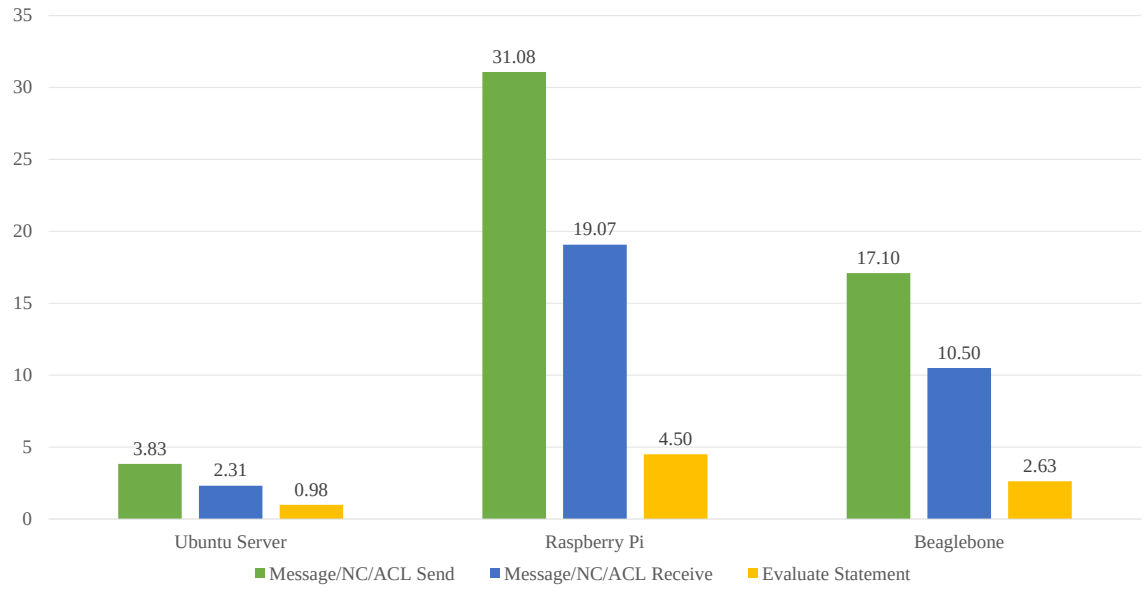

Figure 23: Comparison of UbiPAL send, receive, and evaluate ACL statement in milliseconds. Each data point is the average of over 91, 93, and 80 for send, receive, and evaluate statement, respectively.

overhead of one order of magnitude, even the slowest device in our test array could still send 50 secure, authenticated, and authorized messages per second.

5.2.2 Network Overhead

In a distributed system, network overhead can be important. As the number of services and messages increase, network throughput can suffer. The overhead is computed statically here, then at runtime in an example below.

As discussed above, there are three message types: namespace certificates, access control lists, and messages between services. Namespace certificates and access control lists carry no user data. Assuming an RSA public modulus length of 1024 bits, or 128 bytes, the respective sizes of the two types of messages are shown in Figure 24 and Figure 25. Assuming a standard address length of 15 (address formatted as a string XXX.XXX.XXX.XXX) and a port of 5 digits, the namespace certificate totals to 566 bytes. Assuming four rules of 50 characters each, the

access control list sums to 755 bytes. These values are pure overhead as they carry no actual application data. However, they are a set cost and once a service has received the necessary access control lists and namespace certificates, it needs not receive them again. Furthermore, networks need not be a clique, meaning services need not communicate these messages with all other services in a network.

Message overhead is shown in Figure 26. The overhead of the message is the combination of the header and the signature. Together, the overhead of a message is 407 bytes, excluding only the raw bytes of the message and argument. Although this overhead is set regardless of the message content, there are certain steps that can be taken to reduce the number of bytes sent as overhead. Explicitly stating the identifier of the receiving service is only necessary during the handshake and exchange of AES keys or in unencrypted unicast messages. This allows the receiving service to verify it is indeed the destination of the message. Once regular AES encrypted communication commences, the full identifier is unnecessary as the receiving service must be the intended recipient in order to decrypt the message. The current UbiPAL library implementation truncates to just the first five characters of the identifier as a known value to enable detection of encryption in messages as well as a method to double check the intended recipient. Additionally, the signature, the primary function of which is to authenticate the sender, is unnecessary after AES keys have been exchanged as the sender's identity is confirmed by use of the AES encryption alone, assuming the AES key and IV has stayed secure. This reduces the message overhead to 153 bytes, which is only 37.6% of the original network overhead.

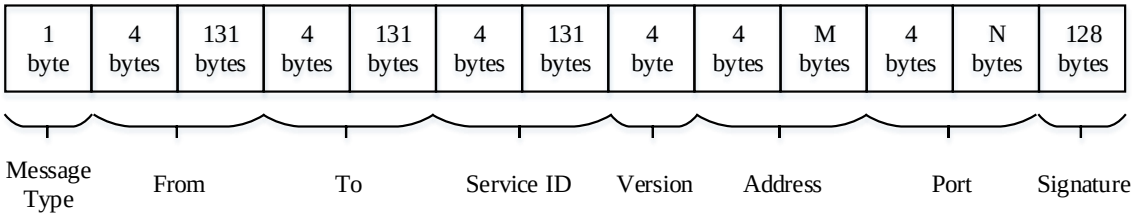

Figure 24: Network byte layout of a UbiPAL namespace certificate.

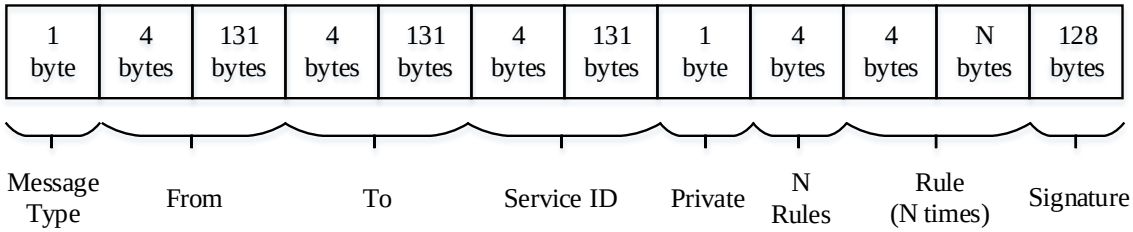

Figure 25: Network byte layout of a UbiPAL access control list.

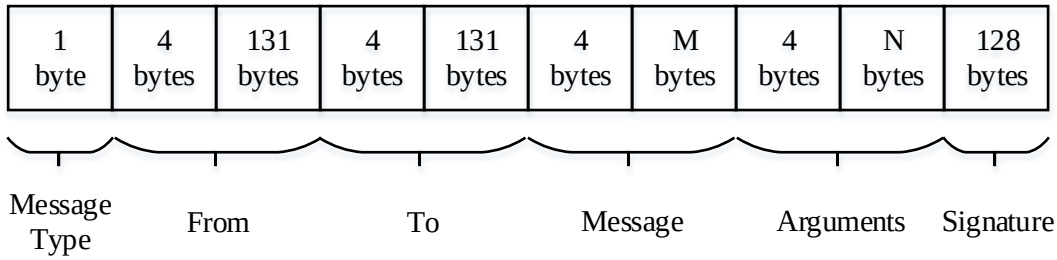

Figure 26: Network byte layout of a UbiPAL message.

However, the simplest way to reduce message overhead is to avoid sending messages altogether. This is done in a few simple ways: message response caching and registration, and access control list and namespace control list requests.

Message response caching, or message registration, is a polling versus interrupt tradeoff. The specific model of UbiPAL message registration is discussed in section 4.3. This is useful in a scenario when a service is requesting data at a higher rate than the data on a remote service is changing. In this case, registering for updates and caching the responses allows the requesting service to store the response message locally and simply deliver the response from the local cache, thus avoiding network requests. The remote service now has the responsibility to send an

update message when its reply message would change, generally when the data it is sharing changes. The tradeoff between polling and pushing can be seen by the following pair of equations:

Equation 1 $$Messages\ when\ polling = 2 * frequency_{poll} * time$$

Equation 2 $$Messages\ with\ pushed\ updates = 2 + (time * \ frequency_{updates})$$

Equation 1 gives the number of messages over time when polling. Polling requires two messages, a message requesting data and its response, multiplied by the frequency of those messages and the span of time being considered. Polling once a second for 10 seconds produces 20 messages. In contrast, the number of messages with pushed updates when registered for a message is described in Equation 2. Two messages are exchanged to register and cache the current reply, then a message is sent on every update of the data source for the time span considered. With an update frequency of every second for 10 seconds, 12 messages are sent. This shows a gain of about 50% to utilize registration rather than standard polling, even when the frequency of updates and the frequency of polling would be the same. Larger gains are received when the update frequency is lower than the polling frequency. As can be seen through simple algebra, network overhead is smaller if the polling frequency is less than half the frequency update time, although data may not be current at all times at the polling service.

Similar computations can be run on the service discovery procedures. One of two methods can be used in a system running UbiPAL, both of which are provided in the C++ UbiPAL library: announce and request. Announcing requires services to broadcast namespace certificates and access control lists on a set interval. Any services hoping to discover other services around them simply wait until the next interval and receive namespace certificates or access control lists that

are sent. Alternatively, a service wishing to discover neighboring services may broadcast requests for either namespace certificates or access control lists from specific services. The UbiPAL library handles replying with namespace certificates or the appropriate non-private access control lists without the application programmer having to define message handlers. Using the request discovery style can save the number of announcement messages which grows linearly with the number of services in the network multiplied by the time span considered.

6. Future Work

Desired future work for UbiPAL includes expansion of mobile platforms supported. Currently, UbiPAL has been implemented as a C++ library for development and evaluation. We believe the mobile and ubiquitous communities stand to gain from the simple access control and communication model presented by UbiPAL. This would be supported by implementation of UbiPAL for various mobile platforms, such as the Android operating system. Other possible platforms may be considered in the future, although the ability to link to existing C++ libraries from many languages allows high reuse of the existing library code with expansion to new device families.

Further work may also be consider on running UbiPAL on extremely low-powered devices such as sensor networks and wearable technology. Although we envision on a small scale the ability to use gateway devices to run UbiPAL on behalf of devices which cannot do so themselves, in a model similar to the home address gateway presented in Mobile IP [15], this does not allow easy scalability to larger networks such as environmental sensor networks. Some approaches utilized in 6LoWPAN [16] or TinyPK [17] may be useful for approaching such resource-constrained devices.

Additional plans also include full-system namespace lookup techniques or access control list gathering. This would be useful in a scenario when services are separated by subnetworks or unavailability yet have network traffic to exchange. Currently, an extended lookup process is left to the devices of the application programmer, but such discovery techniques may be beneficial to the UbiPAL model. Due to the ad hoc nature of UbiPAL systems, lookup techniques could be similar to mesh networking techniques described in papers such as [18], [19], [20].

Finally, we believe it would be interesting to create visual tools for editing and debugging UbiPAL access control lists. Although individual statements are human-readable, reasoning about an entire distributed system can be difficult. A service which could graph the access control tree as discussed above and show the groups of devices on a network which would be granted or disallowed access through a given policy change could be an extremely valuable tool during policy creation on complex UbiPAL systems.

7. Related Works

The access control language of this work is derivative of SecPAL [2], a project from Microsoft Research involving many researchers with publications spanning from 2006 to 2014. SecPAL has been discussed at length in this paper. All referenced SecPAL papers as well as sample projects and other related content can be found at Microsoft Research's SecPAL page at [21]. Similar work to SecPAL and UbiPAL can be found in work by Blaze, Feigenbaum, and Lacy [10]. Their research describes a system in which specific nodes in the system hold given authorities. Any node wishing to act against some resource must collect the signature of a node with the authority over that resource and presents the collection of signatures to the resource guard, which is local to the resource. All names in their system are simply the node's public key, removing a layer of indirection by directly mapping public keys to access control authority. SecPAL and UbiPAL borrow the ideas of localized resource guards and using public keys for names but differ in the application of access control as gathering tokens versus distributing lists.

The UbiPAL system is related to many existing systems. Object-orientation and communication based on network messages, two concepts relied upon heavily by the UbiPAL system, have been used for decades by projects such as the Mach microkernel [22]. Mach is introduced as an object-oriented system allowing programmers to reason about processes as objects communicating through messages, much like modern day object-oriented languages. UbiPAL system also holds some similarities to the Plan B system [23]. Plan B is a ubiquitous system based on Bell Lab's Plan 9 [24] in which all system resources are exported to the network and combined to form a system including the remove resources. The public key infrastructure mirrors some traits of TLS/SSL [8, 9] as well as parts of the Kerberos authentication service [4].

We believe UbiPAL would benefit from some of the capabilities described in other systems. Systems such as instant matchmaking [25] encourage the ability to discover devices immediately around the user that perform certain tasks. A full realization of this may be along the lines of the personal server [26] in which a user carries a small device for computation which utilizes the devices around it. We believe UbiPAL could be a natural fit for such works.

Similar protocols include DCAC [27], which is a recent system that approaches access control on a system in a distributed manner. Each resource has a principle user, who may delegate access to that resource. Principle names are assigned in a hierarchical manner and stored in files on the system in a consistent way that allows any machine mounting that file system to apply the DCAC access control policies in a consistent way. Some work in trust negotiation has followed similar patterns as the UbiPAL system. The authors of [28] describe a system in which devices exchange access control policies and identifying certificates upon a request being made between agents in the system. Credentials are based on the identifying certificates and allow the two agents to agree upon access control policies through mutual negotiation techniques.

There is existing work on making asymmetric cryptography and public key architecture more accessible to the extreme low end of computing devices, such as those often found in sensor networks. This work could be extremely beneficial to any effort to run UbiPAL directly on a sensor network. Work by Watro et al. on TinyPK [17] and Malan et al. [29] are two examples of such work to reduce the overhead of such expensive cryptographic techniques. There is also ongoing work, such as [16], to decrease the overhead of protocols like the Internet Protocol Version 6 [30] to better serve devices with very constrained resources.

Ubiquitous computing can be a very personal computing experience. Devices often may reflect activities users take in the privacy of their own home. According to research by

Srinivasan, Stankovic, and Whitehouse [31], snooping attacks need not even be able to read messages in order to learn about the users' activities inside of a home noticing that correlations between network traffic and specific events can be built relatively easily. Srinivasan et al. present potential methods to avoid such snooping attacks from gaining personal knowledge user.

8. Conclusion

The UbiPAL system successfully completely decentralizes the SecPAL system and allows for authentication and authorization in ad hoc networks of services while protecting against a single point of failure. Combined the with UbiPAL language, defined as extensions of the SecPAL language, the UbiPAL system successfully contextualizes SecPAL and provides an appropriate model for the ubiquitous computing world. The majority of complexity is handled by the implemented library and application programmers must simply write message event handlers and human readable access control policies. The overhead of delivering messages and enforcing access control is kept to a minimum through sparing use of asymmetric cryptography and reduction in UbiPAL header overhead between familiar services. Even on the slowest devices tested, message sending in UbiPAL shows a computational overhead of under 20 milliseconds with 153 bytes of message network overhead. In exchange for this overhead, UbiPAL provides a simple programming model for access control and secure communication through ad hoc authentication and authorization.

Works Cited

[1] C. Bielstein, "UbiPAL on Github," 2015. [Online]. Available: http://www.github.com/CBielstein/UbiPAL.

[2] M. Y. Becker, C. Fournet and A. D. Gordon, "Design and Semantics of a Decentralized Authorization Language," in *IEEE Computer Security Foundations Symposium - CSF*, Venice, 2007.

[3] A. Grünbacher, "POSIX Access Control Lists on Linux," in *USENIX Technical Conference - USENIX*, San Antonio, 2003.

[4] J. Steiner, C. Neuman and J. Schiller, "Kerberos: An authentication service for open network systems.," in *Proceedings of USENIX*, Dallas, 1988.

[5] L. Zhu and B. Tung, "Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)," Internet Engineering Task Force (IETF), 2006.

[6] M. Y. Becker, "SecPAL: Formalization and Extensions," Microsoft Research, 2009.

[7] M. Y. Becker, J. F. Mackay and B. Dillaway, "Abductive Authorization Credential Gathering," in *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, London, 2009.

[8] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," Internet Engineering Task Force (IETF), 1999.

[9] A. Freier, P. Karlton and P. Kocher, "The SSL Protocol Version 3.0," Internet Engineering Task Force (IEFT), 2011.

[10] M. Blaze, J. Feigenbaum and J. Lacy, "Decentralized Trust Management," in *IEEE Symposium on Security and Privacy*, Oakland, 1996.

[11] C. Ellison, "SPKI Requirements," Internet Engineering Task Force (IETF), 1999.

[12] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas and T. Ylonen, "SPKI Certificate Theory," Internet Engineering Task Force (IETF), 1999.

[13] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," Internet Engineering Task Force (IETF), 2003.

[14] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard (AES)," National Institute of Standards and Technology Computer Security Resource Center, 2001.

[15] C. E. Perkins, "Mobile Networking Through Mobile IP," *IEEE Internet Computing - INTERNET*, vol. 2, no. 1, pp. 58-69, 1998.

[16] G. Mulligan, "The 6LoWPAN Architecture," in *Proceedings of the 4th workshop on Embedded networked sensors (EmNets '07)*, 2007.

[17] R. Watro, D. Kong, S. F. Cuti, C. Gardiner, C. Lynn and P. Kruus, "TinyPK: Securing Sensor Networks with Public Key Technology," in *Proceedings of the 2nd ACM WOrkshop on Security of Ad Hoc and Sensor Networks*, Washington, 2004.

[18] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in *IEEE Transactions on Mobile Computing*, 1999.

[19] C. E. Perkins, "Ad Hoc On-Demand Distance Vector Routing," in *Workshop on Mobile COmputing Systems and Applications*, 1999.

[20] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *Computer Communication Review - CCR*, vol. 24, no. 4, pp. 234-244, 1994.

[21] Microsoft Research, "SecPAL," Microsoft Rsearch, [Online]. Available: http://research.microsoft.com/en-us/projects/SecPAL/.

[22] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Technical Conference - USENIX*, 1986.

[23] F. J. Ballesteros, E. Soriano, K. Leal and G. Guardiola, "Plan B: An Operating System for Ubiquitous Computing Environments," in *IEEE International Conference on Pervasive Computing and Communications - PerComm*, Pisa, 2006.

[24] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom, "Plan 9 from Bell Labs," in *In Proceedings of the SUmmer 1990 UKUUG Conference*, 1990.

[25] D. K. Smetters, D. Balfanz, G. Durfee, T. F. Smith and K.-h. Lee, "Instant Matchmaking: Simple and Secure Integrated Ubiquitous Computing Environments," in *Ubiquitous*

Computing/Handheld and Ubiquitous Computing - UbiComp (HUC), Orange County, 2006.

[26] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar and J. Light, "The Personal Server: Changing the Way We Think About Ubiquitous Computing," in *Ubiquitous Computing / Handheld and Ubiquitous Computing - UbiComp (HUC)*, Gothenburg, 2002.

[27] Y. Xu, A. M. Dunn, O. S. Hofmann, M. Z. Lee and E. Witchel, "Application-Defined Decentralized Access Control," in *USENIX Annuall Technical Conference (ATC)*, Philadelphia, 2014.

[28] D. Zou, J. H. Park, L. T. Yang, Z. Liao and T.-h. Kim, "A Formal Framework for Expressing Trust Negotiation in the Ubiquitous Computing Environment," in *5th International Conference for Ubiquitous Intelligence and Computing Proceedings*, Oslo, 2008.

[29] D. Malan, M. Welsh and M. Smith, "A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography," in *First IEEE International Conference on Sensor and Ad Hoc Communications and Network*, Fort Lauderdale, 2004.

[30] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Internet Engineering Task Force (IETF), 1998.

[31] V. Srinivasan, J. Stankovic and K. Whitehouse, "Protecting your daily in-home activity information from a wireless snooping attack," in *Proceedings of the 10th International Conference on Ubiquitous Computing - UbiComp '08*, Seoul, 2008.

[32] H. M. Levy, "Issues in Capability-Based Architectures," in *Capability-Based Computer Systems*, Bedford, Massachusetts: Digital Press, 1984, pp. 187-205.

Vita

Cameron Taylor Bielstein was born and raised in Austin, Texas. After completing a homeschool education in 2009, Cameron entered Austin Community College, where he earned the degree of Associate of Science in Computer Science. After two years, Cameron transferred to The University of Texas at Austin and enrolled in the 5-Year Integrated Master's Program in the Department of Computer Science. Upon completion of the program in May 2015, he was awarded both the degrees of Bachelor of Science in Computer Science and Master of Science in Computer Science.

Email address: cameronb@utexas.edu

This thesis was typed by the author.